

Neural Networks (aka Deep Learning)

Kernels:

Remember the kernel trick? We made linear classifiers non-linear by mapping the data implicitly into a fixed (very) high dimensional feature space.

$$x \rightarrow \phi(x) \quad h(x) = \phi(x)^T w + b$$

Neural Networks:

Neural Nets were originally invented by Frank Rosenblatt in 1963 (at Cornell) and take a similar approach.

However, in contrast to kernels, here the mapping is explicit and learned.

Multi-layer Perceptron (MLP): $\phi(x) = \sigma(Ux)$

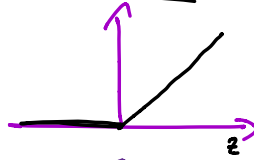
Here $\sigma(z)$ is a non-linear transition function, which operates element-wise on each dimension.

Without the transition function, the classifier would just be linear:

$$w^T \phi(x) + b = v^T(Ux) + b = w^T Ux + b = \underbrace{\tilde{w}^T x + b}_{\text{linear!}} \quad \text{where: } \tilde{w} = v^T U$$

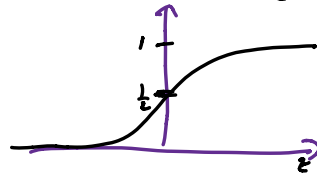
Example transition functions are:

$$\sigma(z) = \max(z, 0)$$



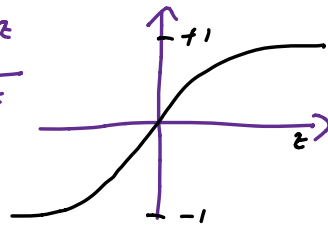
Rectified Linear Unit (ReLU)

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$



Sigmoidal Unit

$$\sigma(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$



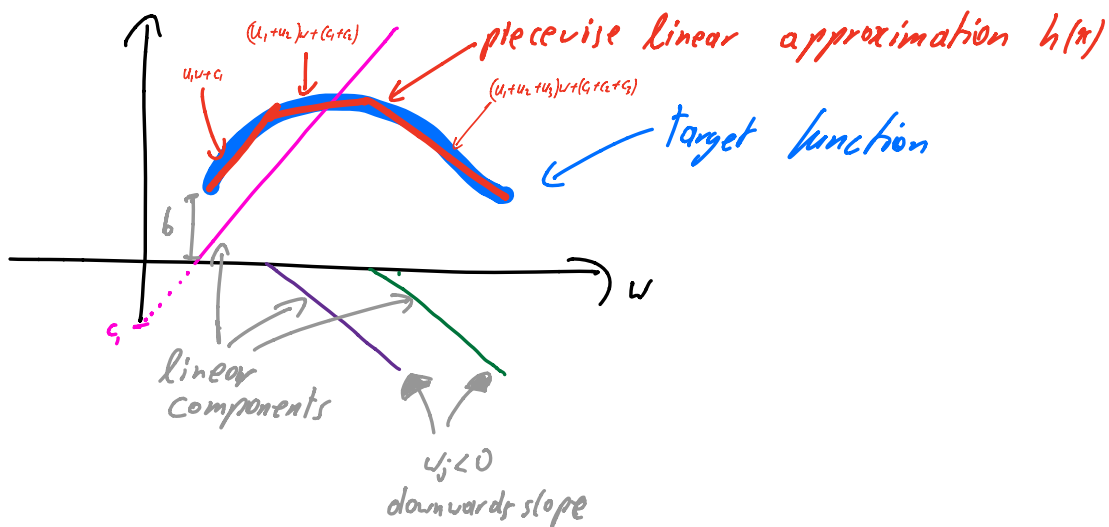
Hyperbolic Tangent

Example (ReLU): (Regression)

$$U = \begin{bmatrix} -u_1^T \\ -u_2^T \\ -u_j^T \end{bmatrix}$$

$$h(x) = w^T \max(Ux + \vec{c}) + b = \sum_j w_j \max(\underbrace{u_j^T w + c_j}_{\text{linear function}}, 0) + b$$

↑ bias



Learning: We need to learn the parameters of ϕ , and we can do simple gradient descent.

Let \mathcal{L} be any well-defined loss function,
 eg. $\mathcal{L}(h) = \frac{1}{2} \sum_{i=1}^n (h(x_i) - y_i)^2$ h has parameters: \vec{w}, u, \vec{c}, b

simple gradient descent:

Repeat

$$\begin{cases} \vec{w} \leftarrow \vec{w} - \alpha \frac{\partial \mathcal{L}}{\partial \vec{w}} \\ u \leftarrow u - \alpha \frac{\partial \mathcal{L}}{\partial u} \\ \vec{c} \leftarrow \vec{c} - \alpha \frac{\partial \mathcal{L}}{\partial \vec{c}} \\ b \leftarrow b - \alpha \frac{\partial \mathcal{L}}{\partial b} \end{cases}$$

These gradients can be computed very efficiently with gradient backpropagation (aka Chain Rule)

Stochastic Gradient Descent (SGD):

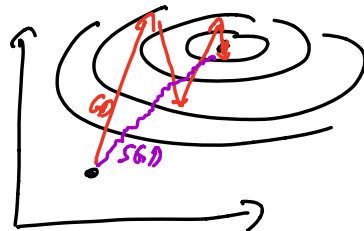
Loss function: $\sum_{i=1}^n \mathcal{L}(h(x_i), y_i)$ eg. $\frac{1}{2} \sum_{i=1}^n (h(x_i) - y_i)^2$

Gradient: $\nabla \mathcal{L} = \sum_{i=1}^n \frac{\partial \mathcal{L}(h(x_i), y_i)}{\partial \mathcal{L}}$

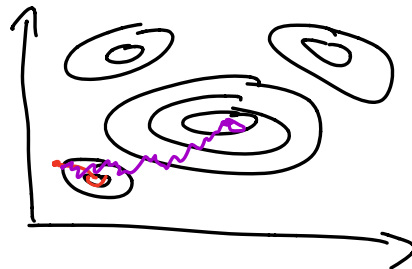
SGD approximates gradient with only 1 or $m \ll n$ samples.

$\nabla \mathcal{L} \approx \frac{\partial \mathcal{L}(h(x_i), y_i)}{\partial \mathcal{L}} \leftarrow$ single x_i

Perform one tiny update for each sample.



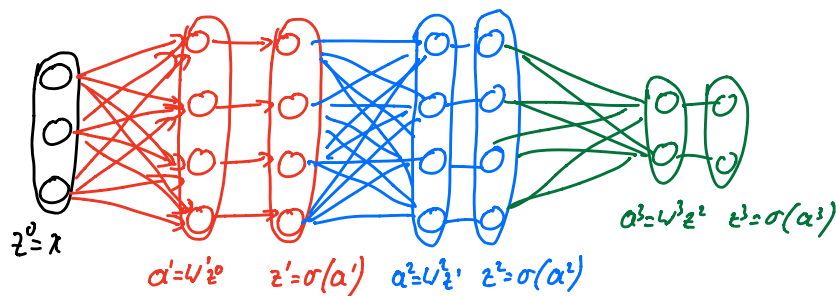
SGD is very noisy. However this turns out to be a crucial "feature" to avoid local minima and saddle points. More precise optimization methods (especially Newton or approximated Newton methods) land in local minima. SGD is "too imprecise" to hit small narrow (bad) local minima.



Deep Networks: One nice property of NN is that we can introduce multiple layers (and increase complexity):

$$\phi(x) = C \sigma(B \sigma(Ax + \vec{a}) + \vec{b}) + \vec{c}$$

Now it gets more complicated. The *first* layer learns linear functions. The *second* layer combines these to multiple non-linear functions. The *third* layer combines multiple of these non-linear functions to even more complex non-linear functions.



The forward propagation is really simple to implement:
Forward pass: (prediction)

$z_0 = x$
 For $l=1:L$

$$a_l = W^l z_{l-1} + b_l$$

$$z_l = \sigma_l(a_l)$$

End

Return z_L

Comments: - Typically the last transition function, σ^L , can differ from the others

↖ bias term in layer l

Amazingly, so is the backwards (gradient) pass:

Backward pass: (Gradient Update)

$$\vec{\delta}_L = \frac{\partial L}{\partial z_L} \odot \sigma'_L(a_L)$$

for $l=L-1:1$

$$W_l = W_l - \alpha \vec{\delta}_l z_{l-1}^T$$

$$b_l = b_l - \alpha \vec{\delta}_l$$

$$\vec{\delta}_{l-1} = \sigma'_{l-1}(a_{l-1}) \odot (W_l^T \vec{\delta}_l)$$

End

\odot is the element wise product

$$\begin{pmatrix} a \\ b \end{pmatrix} \odot \begin{pmatrix} c \\ d \end{pmatrix} = \begin{pmatrix} a \cdot c \\ b \cdot d \end{pmatrix}$$

σ'_l is the gradient of the transition function σ_l

Important optimization tricks:

- Use Momentum
- Reduce stepsize during optimization
- Use mini-batch (SGD) with $m \approx 64$ inputs
- scale features to be within $[0,1]$
- de-correlate features (PCA)
- For image data use Convolutional NN

$$\begin{aligned} G &\leftarrow \beta G + \frac{\partial L}{\partial W} \\ W &\leftarrow W - \alpha G \end{aligned}$$



large step-size
 small step-size

Convolutional Neural Networks (CNN)

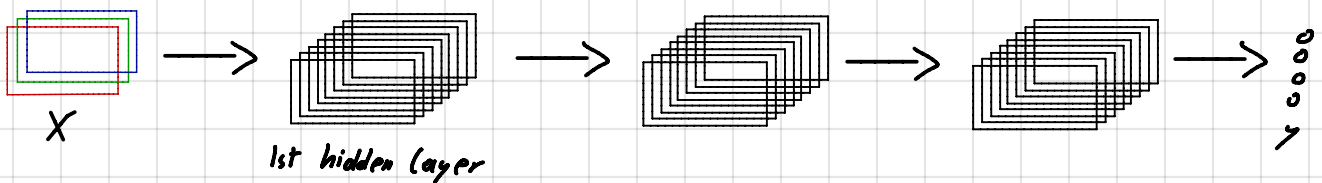
Assume Input is an image. ✓

Size: $W \times H \times 3$



CNNs restrict the linear transformations $X \rightarrow WX$ to be convolutions

These: a) share weights
b) ensure that hidden layers are also images (with many channels)



Convolution operation:

input image

1	3	2	5	7
6	0	1	1	3
2	5	2	7	2

* convolution kernel

1	2
0	1

= output activations

7	7	13	21
11	4	9	9

↳ These are the learned weights

Typically you have one kernel per input channel. The output is the sum of the convolved images.

1	3	2	5	7
6	0	1	1	3
2	5	2	7	2

* convolution kernel

1	2
0	1

=

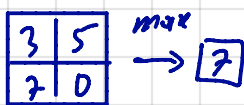
7			
			9

Convolution can be interpreted as a pattern detector. Patches that look similar to the kernel will yield large activations. The conv. operator 'scans' the input for such patterns.

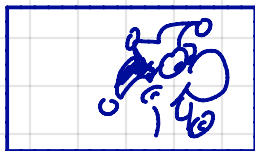
$$7 = 1 \cdot 1 + 3 \cdot 2 + 6 \cdot 0 + 0 \cdot 1$$

$$9 = 1 \cdot 1 + 3 \cdot 2 + 7 \cdot 0 + 2 \cdot 1$$

Pooling: Like convolution, but you reduce a patch to its max or mean value.



Pooling is locally translation invariant.



← These two images yield similar activations.

Batch-Norm: Re-normalize activations based on current mini-batch.

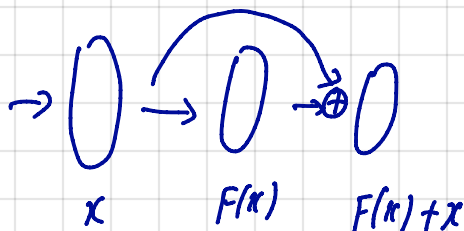
$$a_i \leftarrow \gamma_i \frac{a_i - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}} + \beta_i$$

↑ parameters that are learned

μ_i = mean activation in mini-batch

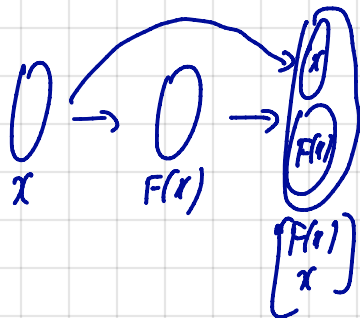
σ_i^2 = variance in mini-batch

Residual Connection:



Add input to output activations after each layer*.

Dense Connection:



Concatenate input to output activations after each layer*.

* typically here layers are [Conv ⊙ BN ⊙ ReLU]